$\mathcal{ABC}$-VHDL

A Synchronous VHDL Subset with a Formal Semantics in HOL

Dirk Eisenbiegler and Ramayya Kumar
Forschungszentrum Informatik
(Prof. Dr.-Ing. D. Schmid)
Haid–und–Neu–Straße 10-14 76131 Karlsruhe, Germany
e–mail: {eisenbiegler,kumar}@fzi.de

November 20, 1995

Abstract

VHDL is frequently used for describing purely synchronous circuits. However, the underlying model of VHDL is much more expressive than it need be. In this report, a synchronous subset of VHDL named $\mathcal{ABC}$-VHDL is introduced. $\mathcal{ABC}$-VHDL is dedicated towards logical argumentation and correct circuit synthesis based on VHDL descriptions. Although being conform with the standard VHDL semantics, the semantics of $\mathcal{ABC}$-VHDL is based on a far simpler model: synchronous circuit descriptions at the RT-level formalized within higher order logic. This article describes the syntactical aspects of $\mathcal{ABC}$-VHDL, and it also defines the semantics of $\mathcal{ABC}$-VHDL by a mapping between $\mathcal{ABC}$-VHDL structures and the corresponding formulae in higher order logic.

Contents

Chapter 1

Motivation

To argue about VHDL programs on the logical level, a formal semantics, i.e. a mapping between the syntactic VHDL structures and the corresponding logical formulae, has to be found. Experiences in this area show, that defining a formal semantics is a sophisticated goal due to the complexity of the language [BGGH92, DaJS93, BrFK94]. The given informal VHDL semantics is not as precise as it should be, and this is why very often different VHDL simulators lead to different simulation results and different formalization approaches lead to different formal semantics. Another problem is the complexity of the underlying timing model. The resulting formulae have to represent this timing model in an adequate manner, and this is why the resulting formulae may become very complex, and verification may become an exacting process.

In the area of hardware synthesis, VHDL is often used to describe nothing but synchronous circuits. Usually the syntax of VHDL sub-languages is defined by sets of restrictions and the semantics is inherited from VHDL [DeOd93]. Although synchronous circuits can easily be formalized in a higher order logic calculus, deriving the semantics of VHDL descriptions in general is rather difficult. The underlying model is much more expressive and powerful than it need be for just describing synchronous circuits.

In this paper, a reduced VHDL language called $\mathcal{ABC}$-VHDL is introduced. $\mathcal{ABC}$-VHDL is restricted to synchronous circuit descriptions. For $\mathcal{ABC}$-VHDL a new timing model has been defined. The $\mathcal{ABC}$-VHDL timing model conforms to the VHDL timing model, but it is tailored for synchronous circuits only, and this makes it far simpler.

VHDL programs do not always represent real circuits in an adequate way. For example, they may have infinite loops, such that a process keeps being executed without ever reaching a wait statement. Another example for inadequate circuit descriptions are structures with zero–delay cycles, i.e. a cycle, where each process produces a delta delay output and passes it to the next process of the cycle. When formalizing circuit descriptions in logic, it is very important to ensure, that the resulting formulae are consistent. Inconsistent formulae never correspond to technically realizable circuits, but any property can formally be derived from them (*ex falso quodlibet*). $\mathcal{ABC}$-VHDL puts a stress on consistency. It is restricted to well defined synchronous circuit descriptions. Zero–delay cycles, infinite loops, short-circuits etc. are avoided.

All processes are described by means of the lambda calculus. Lambda terms are both logical specifications and executable programs. The formulae to be constructed not only formally describe the processes, but can also be used for simulation by evaluating the lambda terms in an interpreter of a functional programming language.

In general, there are two ways of formally embedding languages: deep embedding and shallow embedding [BGGH92, ReKr93b]. Deep embedding means that the syntax of the language is represented by terms within the logic, and formulae are defined to describe the semantics of these logical structures. Shallow embedding means, that in the logic there are no terms representing the syntax of the language. There is just a function (a program) mapping the syntax represented outside logic to a formula within the logic.

We chose a shallow embedding approach. A theory with constants and types for representing $\mathcal{ABC}$-VHDL has been invented within HOL. This theory has been designed in a way, such that $\mathcal{ABC}$-VHDL representations and their corresponding HOL representations are very close to one another, i.e. differ only in minor syntactical aspects. A program for automatically converting $\mathcal{ABC}$-VHDL programs into HOL representations has been implemented.

The paper is structured as follows: Chapter 2 introduces the main principles of $\mathcal{ABC}$-VHDL and the major syntactical restrictions compared with standard VHDL. In the next chapters, the $\mathcal{ABC}$-VHDL language is built bottom up. Chapters 3, 4 and 5 describe the semantics of $\mathcal{ABC}$-VHDLs entities, architectures, processes, and statements. Appendix A enlists the syntax of $\mathcal{ABC}$-VHDL. In appendix B the elements defined in HOL are enlisted.

Chapter 2

$\mathcal{ABC}$-VHDL versus VHDL

2.1 Processes

In $\mathcal{ABC}$-VHDL, only pure input and pure output signals are allowed. Statement parts of processes are compound statements that are recursively constructed using certain basic statements and control structures. As basic statements there are:

1. wait statements,
2. signal assignments,
3. variable assignments and
4. null statements

As control structures there are:

1. sequences of statements,
2. loops and
3. if-then-else structures

General VHDL programs are not restricted to synchronous circuits. In $\mathcal{ABC}$-VHDL, every sequential program and even every atomic or compound statement will be represented by an output and transition function, that maps input and current state to output and next state. In $\mathcal{ABC}$-VHDL, atomic circuits may either have one or zero clock inputs, and all wait statements must have the form

```
wait until clk = '1';
```

where `clk` is the clock signal of this circuit. The clock signal must not be used within variable assignments and signal assignments.

In $\mathcal{ABC}$-VHDL, simulation cycles are clock cycles. A simulation cycle starts with a rising slope of the clock signal. During a clock cycle the processes may read the input signals and the current variables and depending on these values certain variable and signal assignments are executed and finally a new wait statement is reached.

In $\mathcal{ABC}$-VHDL, only zero delay signal assignments are allowed. After getting the positive slope of the clock signal, the process will immediately read the input and it will produce its output signal assignments at this very moment.

Figure 2.1 describes the life of an ABC-VHDL process. The gray shape symbolizes a program: s_0 denotes the beginning state, $\mathsf{s}_1, \mathsf{s}_2, \ldots \mathsf{s}_n$ denote the wait statement occurrences, and the arrows denote transitions. In other words, the total number of control states is equal to the number of wait statements plus one. An arrow from some s_m to some s_n indicates, that from one clock tick to another the process *may* jump from control state s_m to control state s_n. Usually there may be several arrows starting from one control state, since the succeeding control state not only depends on the current control state but also on the current values of the variables and inputs.

The control states $\mathsf{s}_0, \mathsf{s}_1, \mathsf{s}_2, \ldots \mathsf{s}_n$ correspond to the control states of the synchronous circuit that is to be described by this program. At time 0, the process starts in the beginning state s_0. After the first clock tick, it reaches a wait statement, and in all further clock cycles it jumps from one wait statement occurrence to another. The initial state s_0 will never be reached again. Whenever the execution reaches the end of the program, it will automatically proceed at the beginning without delay. When passing through the end and continuing in the beginning the execution will *not* stop at position s_0 to wait there for a clock period.

There are two kinds of control states a process can be in. First there is the initial state s_0, and second there are the control states $\mathsf{s}_1, \mathsf{s}_2, \ldots$ related to the wait statements occurrences. Statements, i.e. parts of ABC-VHDL programs, will be described in the same way as entire programs except that they not only have a beginning s_0 but also an end s' (see figure 2.2). Being in state s_0 means, that the execution of the processes recently passes through the beginning of the statement. Reaching s' means, that the process has finished the execution of the statement and that it will immediately continue with the next statement if there is any. In statements, s_0 and s' do *not* correspond to control states in the corresponding synchronous circuit.

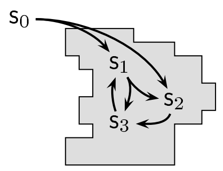


Figure 2.1: A Process

2.2 Architectures

In $\mathcal{ABC}$-VHDL as well as in VHDL, structures are described by means of architectures. $\mathcal{ABC}$-VHDL neither supports generic architectures nor generate statements are. Furthermore, there is restriction on how structures may be built: short-circuits and zero-delay-cycles are not allowed.

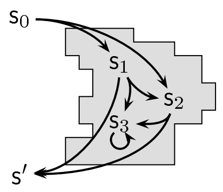


Figure 2.2: A Statement

The use of clock signals is restricted. When there is at least one part of the structure with a clock input, then the entire circuit must have one clock input, that is connected to all clock inputs of the sub-circuits. All clock signals must be interconnected. Clock signals and non-clock signals must not be connected.

2.3 Types, Constants, Functions, Operators and Expressions

$\mathcal{ABC}$-VHDL does not allow type declarations nor does it allow function or constant declarations. However, arbitrary types, functions and constants are allowed. But before they can be used, their semantics have to be defined explicitly by giving a corresponding representation in HOL and by defining a mapping between data types, operators and constants in $\mathcal{ABC}$-VHDL and in HOL. There are two tables: the type table, which defines the semantics of $\mathcal{ABC}$-VHDL data types and the constant table, which defines the semantics of $\mathcal{ABC}$-VHDL constants, functions and operators.

Appendix C gives an example for such mappings. Throughout this paper, these mappings will be used. However, the user may define arbitrary other mappings.

Expressions are built recursively. Constants, variables and input signals are expressions. Compound expressions are built by applying functions to tuples of basic elements or previously built compound expressions. All expressions must be well typed, i.e. a type is assigned to all basic elements and for all functions it is defined what are the types of its inputs and what is the type of the result.

2.4 Summary

The main differences between $\mathcal{ABC}$-VHDL and VHDL are:

- $\mathcal{ABC}$-VHDL is restricted to purely synchronous circuits
- only pure input and pure output signals are allowed
- each circuit may have a single clock input
- short-circuits and zero-delays are prohibited
- there are no generic architectures
- there are no generate statements

- type, constant and function definitions cannot be defined in $\mathcal{ABC}$-VHDL
- the semantics of types, constants and functions used has to be defined explicitly

Chapter 3

Libraries, Entities and Architectures

Throughout this paper, only synchronous circuits at the RT and logical abstraction level will be considered. Time is considered to be discrete, and the clock ticks are formally represented by natural numbers. All combinatorial circuits have zero delay, and D-flipflops have unit delay, i.e. one clock period.

3.1 Libraries

Whenever VHDL source text is elaborated by the parser, it is assigned to a library. The library name is not defined within the VHDL source text but is handled to the parser explicitly by means of user interaction.

In $\mathcal{ABC}$-VHDL, the only elements within libraries are entities and architectures. Previously defined architectures may be used as components within architecture bodies. However, if the library the architecture belongs to differs from the current library, then the library has to be declared by means of a library clause. Library clauses are always related to the next entity or architecture declaration. Libraries declared for entities can be used in all its architectures.

3.2 Entities

In an entity declaration, the entity name and the names and types of its signal interface are defined. Entity declarations describe the interface of architecture, whereas the architecture describes its behavior.

There are two types of signals: data signals and clock signals. Each entity may have one clock signal. The usage signals must not differ within different architectures. It is not allowed to use one signal as clock signal in one architecture and as data signal in another. Within the logical representation, clock signals will not be formalized explicitly (see chapters 4 and 5).

In $\mathcal{ABC}$-VHDL, only pure input and pure output signals are allowed. Let i^1, i^2,… i^{n_i} be the data input signals and o^1, o^2,… o^{n_o} be the output signals, ι^1, ι^2,… ι^{n_i} be the types of the input signals and ω^1, ω^2,… ω^{n_o} be the types of the output signals of the architecture.

In $\mathcal{ABC}$-VHDL, interface signals can be instantiated within the entity declaration. However, due to the restrictions in $\mathcal{ABC}$-VHDL, instantiations of input signals have no effect and therefore are ignored. Using instantiations for interface signals is optional. Whenever an output signal remains uninstantiated, existentially quantified variables are used to represent these values in logic.

Example

The entity declaration in figure 4.1 describes the interface of an entity named gcd. The signal clock is used as clock signal, since in the architecture below it is used within wait-statements.

There are three data input signals named *a*, *b* and *start*. Where *a* and *b* are of type `positive` and *start* is of type `std_logic`. According to appendix C, the $\mathcal{ABC}$-VHDL type `positive` is represented by the HOL type num and `std_logic` is represented by bool. So the type of the input ι becomes

$$\mathsf{num} \times \mathsf{num} \times \mathsf{bool}$$

There are two output signals named *ready* and *result* of type `std_logic` and `positive` and instantiations `'0'` and `0`, respectively. According to appendix C, the $\mathcal{ABC}$-VHDL type `positive` is represented by the HOL type num and `std_logic` is represented by bool. The type of the output ω is

$$\mathsf{num} \times \mathsf{bool}$$

According to appendix C, the expressions `'0'` and `0` are represented by F and 0, respectively. The initial state of the output signal is

$$(\mathsf{F}, 0)$$

3.3 Architectures

All architectures are related to some entity declaration, where its entity name, the input signals and the output signals are defined. Architectures always belong to the same library as their entity declarations. Within the declarative part of architectures, internal signals may be defined and instantiated. Like output signals and unlike input signals, instantiations of internal signals do have an effect on the behavior and are not ignored.

The statement part of an architecture defines a structure. Its parts are named concurrent statements and they are interconnected via signals.

Formal Representation of Architectures

All architectures are represented by relations between time dependent input and output signals. Time dependent signals are represented by functions mapping time to a value of some data type α. An arbitrary type α may be used for the value of a signal. In HOL, time is represented by the data type num (natural number: $0, 1, \ldots$). Signals have the following type:

$$\mathsf{num} \to \alpha$$

Architectures are represented by relations between signals, i.e. by a function mapping a tuple of signals to bool (boolean values: T or F). In HOL, terms representing architectures have the following type:

$$
\begin{array}{l}
(\\
\quad (\mathsf{num} \to \iota^1) \times (\mathsf{num} \to \iota^2) \times \ldots \times (\mathsf{num} \to \iota^{n_i}) \ \times \\
\quad (\mathsf{num} \to \omega^1) \times (\mathsf{num} \to \omega^2) \times \ldots \times (\mathsf{num} \to \omega^{n_i}) \\
) \\
\ \to \mathsf{bool}
\end{array}
$$

This type only depends on the entity declaration. All architectures of the same entity are represented by relations with the same type.

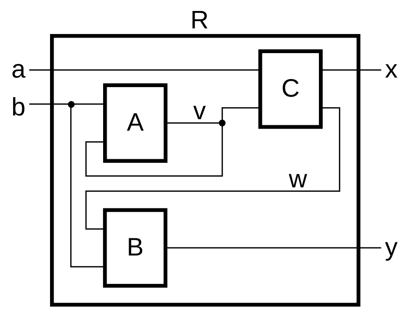
Structures

Representing structures in higher order logic is straightforward [HaDa86]. The general scheme is as follows:

$$
\begin{array}{l}
\forall i^1, i^2, \ldots i^{n_i}, o^1, o^2, \ldots o^{n_o}. \\
\quad \exists y^1, x^2, \ldots y^m. \\
\quad\quad R(x^1, x^2, \ldots x^n) = R^1(\ldots) \wedge R^2(\ldots) \wedge \ldots \wedge R^k(\ldots)
\end{array}
$$

In this formula, a compound circuit R is defined as a composition of its parts $R^1, R^2, \ldots R^k$. The external signals $i^1, i^2, \ldots i^{n_i}, o^1, o^2, \ldots o^{n_o}$ are all-quantified and the internal signals $y^1, y^2, \ldots y^m$ are existentially quantified. The interface of the compound circuit is connected with all external signals, the interfaces of its components may be connected to arbitrary internal or external signals according to the given net list. In such net list descriptions, circuits are represented by relations. Input and output signals are not distinguished, and there may be several input and output signals.

Figure 3.1 gives an example for a formalization of a circuit structure according to this scheme. The compound circuit is named R, its parts are named A, B and C. The inputs of R are a and b, its outputs are x and y and there are two internal lines named v and w.



$$
\begin{array}{l}
\forall a, b, x, y. \\
\quad \exists v, w. \\
\quad\quad R(a, b, x, y) = A(a, v, v) \wedge B(w, b, y) \wedge C(a, v, x, w)
\end{array}
$$

Figure 3.1: Formalization of a Structure

Concurrent Statements

The structure described by an architecture consists of several concurrent statements. In $\mathcal{ABC}$-VHDL, the following concurrent statements are allowed:

1. processes statements
2. concurrent signal assignments
3. component instantiation statements

According to our formalization scheme, each concurrent statement has to be represented by a relation between time dependent input and output signals.

The logical formalization of processes will be considered in chapter 4. Concurrent signal assignments are equivalent to processes with exactly one sequential signal assignment. Therefore, their semantics is not considered explicitly.

Component instantiations are used to instantiate another architecture as a concurrent statement within the current architecture. Before components can be used, they have to be declared in the declarative part of the architecture. The component declaration a component name is defined and this component name is related to some architecture, which is given by its library, entity and architecture name. Furthermore, the component declaration defines a signal interface. The types and names of the signal interface of the component must equal the types and names of the signal interface of the architecture the component is related to.

There may also be signal instantiations within interface lists of component declarations. So there are two ways to instantiate interface signals: they can be instantiated within the entity declaration and they can also be instantiated within the component declaration, where an architecture of the entity is instantiated. If some signal is instantiated at both places, then the signal instantiation within the component instantiation dominates.

All concurrent statements may have labels. Arbitrary identifiers may be used as labels. All labels used within one architecture must differ. In $\mathcal{ABC}$-VHDL, labels only have an effect to process statements. Different component instantiations with the same component name may have different component instantiations. The component declaration can be restricted to some explicitly enumerated labels.

Within a component instantiation, the association between the signals of the current architecture (actual signals) and the signals of the component (formal signals) is described by the port map. The association between actual signals and formal signals can be done in two ways:

1. a tuple of actual signals, where the order of the elements corresponds to the order of formal signals given in the component specification
2. a tuple, where each actual signal is assigned to one formal signal and the order is ambiguous

In logic, port maps are always represented by tuples. In order to derive the formal representation, port maps of the second group have to be rearranged according to the interface list of the component declaration.

Restrictions to Structures

Structural descriptions at the synchronous abstraction level are not always consistent. In $\mathcal{ABC}$-VHDL, structural descriptions within the architecture body are allowed only if they meet the following restrictions

1. no short-circuits: output signals of components and input signals of the compound circuit must not be connected with other output signals of components or input signals of the compound circuit

2. no zero-delay-cycles: there must not be no ring of combinatorial (zero-delay) circuits, such that one of the outputs of each of them is connected with some input of its successor

3. separated clock: clock signals must not be connected with data signals, and all clock signals must interconnected

In $\mathcal{ABC}$-VHDL, input and output signals are always clearly distinguished. Detecting short-circuits is pretty easy.

To detect zero-delay cycles, zero-delay-dependencies are determined for all processes and architectures. The signal-delay-dependency list is a table with all outputs and the inputs that have a direct impact on this output. A direct impact means, that there is a pure combinatorial path from input to output with no memory unit (delta delay unit) in between.

All basic circuits are defined by means of a process statement or a concurrent signal assignment statement. There are two kinds of processes: ones where all outputs (may) directly depend on all inputs and others where there is no output that directly depends on an input (see sections 4 and 5). To determine the dependencies of architectures, first the dependency lists of its parts have to be determined. Determining the dependency list of an architecture fails if the structure contains a zero-delay-cycle.

Chapter 4

Processes

4.1 The Process Interface

In VHDL, processes are always part of the architecture body. Other than architectures, there is no explicit interface definition for processes. The only signals a processes may use, are the interface signals of its entity declaration plus the signals declared in the declarative part of the architecture. The names, types and instantiations of the signals are defined there.

Although processes do not have an explicit interface declaration, it can be derived. All signals appearing in expressions are called input signals of the process. All signals occurring at some left hand side of a signal assignment are called output signals. In $\mathcal{ABC}$-VHDL, input and output signals must be disjunct. Let $i^1, i^2, \ldots i^{n_i}$ be the input signals and $o^1, o^2, \ldots o^{n_o}$ be the output signals, $\iota^1, \iota^2, \ldots \iota^{n_i}$ be the types of the input signals and $\omega^1, \omega^2, \ldots \omega^{n_o}$ be the types of the output signals of the process. Let ι be an abbreviation for $\iota^1 \times \iota^2 \times \ldots \times \iota^{n_i}$ and ω be an abbreviation for $\omega^1 \times \omega^2 \times \ldots \times \omega^{n_i}$.

4.2 The State of a Process

In the declarative part of processes, variables may be declared and instantiated. Let $v^1, v^2, \ldots v^{n_v}$ be the variables declared in the declaration part of the process, and let $\phi^1, \phi^2, \ldots \phi^{n_v}$ be the corresponding types. Let γ be the type of the control state. γ depends on the structure of the algorithm. This topic will be discussed in chapter 5.

The state of the process σ consists of the following parts: the variable state, the output state and the control state. In $\mathcal{ABC}$-VHDL, all processes have a variable state and an output state. There are two kinds of processes: ones with and others without control state. So σ becomes either $\phi \times \omega \times \gamma$ or $\phi \times \omega$.

4.3 Behavior of Processes

The behavior of processes is determined by its statement part and the instantiation given to the process state. The initial state q is given by the instantiations of the variables, output signals and the initial control state. Its type is σ. The

```
entity gcd is
  port (
    clk    : in std_logic;
    a,b    : in positive;
    start  : in std_logic;
    ready  : out std_logic := '0';
    result : out positive  := 0
  );
end gcd;

architecture behavior of gcd is
  begin process
    variable x,y,z : positive := 0;
  begin
    while start /= '1' loop
      wait until clk = '1';
    end loop;
    ready <= '0';
    if (a < b) then
      x := b;
      y := a;
    else
      x := a;
      y := b;
    end if;
    while (y /= 0) loop
      z := x - y;
      wait until clk = '1';
      x := y;
      y := z;
    end loop;
    ready <= '1';
    result <= x;
    wait until clk = '1';
  end process;
end behavior;
```

Figure 4.1: GCD circuit description in $\mathcal{ABC}$-VHDL

process body can unambiguously be represented by a compound output and transition function, that maps the input (type ι) and the current state (type σ) at time n to the output (type ω) at time n and the state (type σ) at time $n+1$. The process body is represented by a function f having type $\iota \times \sigma \rightarrow \omega \times \sigma$. How f is derived from the statement part of an $\mathcal{ABC}$-VHDL process will be discussed in chapter 5.

f and q determine the process behavior in an unambiguous manner. To derive a relation between the input and output signals of the process the HOL function automaton will be used. automaton is part of the HOL theory Automata (see [EiKu95]). It maps (f,q) with type

$$(\iota \times \sigma \rightarrow \omega \times \sigma) \times \sigma$$

to a function $g = \mathsf{automaton}(f,q)$ with type

$$(\mathsf{num} \rightarrow \iota) \rightarrow (\mathsf{num} \rightarrow \omega)$$

g is a function that maps a time dependent input signal to a type dependent output signal.

The function automaton was been defined by means of induction over time (natural numbers). At time 0, the state is q. f maps the input signal and the state at some time t to the output at time t and the state at time $t+1$. Figure 4.2 sketches the semantics that was given to automaton. $\mathsf{automaton}(f,q)$ can be considered to consist of a combinatorial unit f and a memory unit $\mathcal{D}^q$ with initial value q.

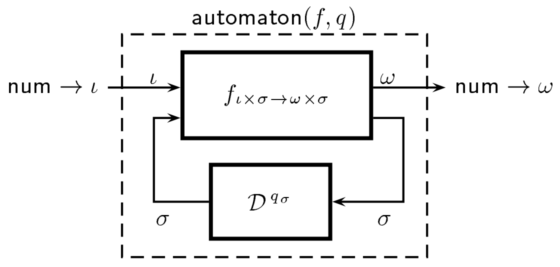


Figure 4.2: Automaton

According to chapter 3, a process is represented by some relation between its input and output signals. $\mathsf{automaton}(f,q)$ is a function rather than a relation, and the input signals and output signals are bundled. To achieve the required relation, an equation has to be built, and the bundled input and output signals have to be split into their parts. The scheme for representing processes is as follows:

$$(\lambda t.(s^1(t),\ldots,s^p(t))) = \mathsf{automaton}(f,q)\ (\lambda t.(s^{p+1}(t),\ldots,s^q(t))$$

Chapter 5

Statements

The body of a process consists of a statement part. As already mentioned in chapter 4, the entire statement part is represented by an output and state transition function f of type $\iota \times \sigma \rightarrow \omega \times \sigma$. We will first describe the formal representation of atomic and compound statements and then describe, how f is derived from the the statement part, i.e. a compound statement.

The logical type and constant definitions used in this chapter are enlisted in appendix B.

5.1 Data Types Used

The following five HOL data types will be used in this chapter: bool, one, num, $(\alpha)\mathsf{option}$ and $\alpha + \beta$. Table 5.1 gives an informal definition of these datatypes. bool represents a set consisting of the elements T and F. The datatype one represents a set that only contains a unique element. The name one is used for both: for the name of the type and also for the constant representing its only element. num represents the natural numbers. It is defined in a recursive manner. There are two constructors: the constant 0 of type num and the constant SUC, a function mapping type num to num. The natural numbers are defined as the set of all expressions built up by these two constructors, i.e. 0, $\mathsf{SUC}\,0$, $\mathsf{SUC}(\mathsf{SUC}\,0)$, $\ldots$.

option and $+$ are type operators. option is a unary type operator that maps an arbitrary type α to a data type named $(\alpha)\mathsf{option}$, where $(\alpha)\mathsf{option}$ represents a set consisting of the element none and of the elements $\mathsf{any}(x)$ for all x of type α. $+$ is a binary type operator, that is used in infix style. It maps two arbitrary types α and β to $\alpha + \beta$, where $\alpha + \beta$ represents a set consisting of the elements $\mathsf{INL}(x)$ for every x of type α and the elements $\mathsf{INR}(y)$ for every y of type β.

5.2 Type $\mathcal{A}$, Type $\mathcal{B}$, and Type $\mathcal{C}$ Statements

In order to formalize sequential statements of $\mathcal{ABC}$-VHDL, we distinguish three classes named $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$. In simplified terms, the differences as follows: On their way from position s_0 to s', the type $\mathcal{A}$ statement *never*, the type $\mathcal{B}$ statement *sometimes* and the type $\mathcal{C}$ statement *always* reaches a wait statement (see figure 5.1).

bool	=	T | F
one	=	one
num	=	0 | SUC *of* num
(α)option	=	none | any *of* α
$\alpha + \beta$	=	INL *of* α | INR *of* β

Table 5.1: Data Types

Type $\mathcal{A}$ statements contain no wait statements. The complete execution of a type $\mathcal{A}$ statement is always performed within a single simulation cycle. Type $\mathcal{B}$ and type $\mathcal{C}$ statements both do contain at least one wait statement. For type $\mathcal{C}$ statements the following property must be fulfilled: Starting at the beginning s_0 the execution will always reach a wait statement before reaching the end s'. For type $\mathcal{B}$ statements this need not be guaranteed. Type $\mathcal{B}$ statements are more general than type $\mathcal{C}$ statements. Type $\mathcal{C}$ statements always are type $\mathcal{B}$ statements but not the other way round.

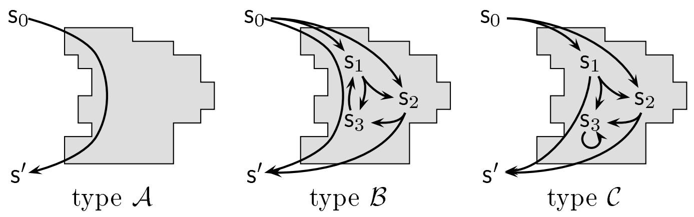


Figure 5.1: Classification of Statements

The logical representations of $\mathcal{ABC}$-VHDL statements define, how the statement affects the state of the process and what output is produced according to the current process state and the current input. There are two reasons for classifying sequential statements: First these three classes of statements will each have a different logical representation, and second there will be restrictions in combining these three kinds of statements in order to avoid nonterminating programs.

Let ι be the type of the input of the process, ω be the type of the output of the process and ϕ be the type of the variables of the process.

Type $\mathcal{A}$ Statements

Type $\mathcal{A}$ statements are represented by functions mapping the input, the old output and the old variable state to the current output and the new variable state. Type $\mathcal{A}$ statements have the following type:

$$\iota \times \omega \times \phi \rightarrow \omega \times \phi$$

Type $\mathcal{B}$ Statements

Type $\mathcal{B}$ statements are represented by functions mapping the old input, the old output, the variable state and the control state to the current output and the new variable and control state. The logical type of type $\mathcal{B}$ statements is defined as follows:

$$\iota \times \omega \times \phi \times (\gamma)\mathsf{option} \rightarrow \omega \times \phi \times (\gamma)\mathsf{option}$$

Type γ is used to represent the set of wait statement positions within the statement. It depends on how the statement is built (see section 5.4). All other types (ι, ω and ϕ) only depend on the process the statement is in.

The control state the processor comes from is either one of the wait statement positions or the beginning of the statement. By applying the type oparator option to γ, one element is added. The elements $\mathsf{any}(x)$, where x is some element of type γ are used to indicate wait statement positions. The extra element none is used to indicate the beginning.

The control state the processor goes to is either one of the wait statement positions or the end of the statement. Also this set of control states is represented by option. The elements $\mathsf{any}(x)$ are again used to indicate wait statement positions, and the extra element none is used to indicate the end of the statement.

Type $\mathcal{C}$ Statements

Type $\mathcal{C}$ statements are formalized similar to type $\mathcal{B}$ statements except that the function is split into a pair of two functions. The first function describes the behaviour when being in the initial state b. Due to the definition of type $\mathcal{C}$ statements, starting from the beginning, the next control state can only be a wait statement position. The first function has the following type:

$$\iota \times \omega \times \phi \rightarrow o \times \phi \times \gamma.$$

The second function describes the behavior starting from some wait statement within the statement. Its type is as follows:

$$\iota \times \omega \times \phi \times \gamma \rightarrow \omega \times \phi \times (\gamma)\mathsf{option}$$

Type $\mathcal{C}$ statements have the following logical type:

$$\begin{aligned}&(\iota \times \omega \times \phi \rightarrow \omega \times \phi \times \gamma) \quad \times \\ &(\iota \times \omega \times \phi \times \gamma \rightarrow \omega \times \phi \times (\gamma)\mathsf{option})\end{aligned}$$

Conditions

Conditions are used within if-then-else structures and loops. They are represented by predicates on input signals and variable states. Their type is as follows:

$$\iota \times \phi \rightarrow \mathsf{bool}$$

Abbreviation	Original Type
(ι, ω, ϕ)type_a_statement	$\iota \times \omega \times \phi \rightarrow \omega \times \phi$
$(\iota, \omega, \phi, \gamma)$type_b_statement	$\iota \times \omega \times \phi \times (\gamma)$option $\rightarrow \omega \times \phi \times (\gamma)$option
$(\iota, \omega, \phi, \gamma)$type_c_statement	$(\iota \times \omega \times \phi \rightarrow \omega \times \phi \times \gamma) \times$ $(\iota \times \omega \times \phi \times \gamma \rightarrow \omega \times \phi \times (\gamma)$option$)$
(ι, ϕ)condition	$\iota \times \phi \rightarrow$ bool

Table 5.2: Type Abbreviatios

Type Abbreviations for Statements and Conditions

In order to simplify the handling with statements and conditions, type abbreviations will be used. Figure 5.2 enlists the type abbreviations and the corresponding types. To switch some term of the original type representation to a term with the new type representation, mk_type_a_statement, mk_type_b_statement, mk_type_c_statement, and mk_condition is applied, and dest_type_a_statement, dest_type_b_statement, dest_type_c_statement, and dest_condition is applied to switch back. The logical definitions of these type abbreviations are enlisted in appendix B.2.

5.3 Basic Statements

In $\mathcal{ABC}$-VHDL, there are four basic statements:

1. signal assignments,
2. variable assignments,
3. null statements, and
4. wait statements

Variable assignments, signal assignments and null are type $\mathcal{A}$ statements. Wait statements are type $\mathcal{C}$ statements.

Signal Assignments

A signal assignment replaces the value of one output signal by the expression given on the right hand side of the signal assignment. Variables are not affected by signal assignments, and the expression on the right hand side only depends on variables and input signals but not of output signals. Therefore signal assignments can unambiguously be described by a function g of type:

$$\iota \times \omega \times \phi \rightarrow \omega$$

Since signal assignemts alter only one of the output signals, the output of g equals its output state input, except that exactly one signal is changed.

The function $\mathsf{SIGNAL\_ASSIGNMENT}$ maps such functions g into an expression of type $\mathsf{type\_a\_statement}$. The new output state is determined by g and the variable state v is left unchangeged.

$$\vdash \mathsf{SIGNAL\_ASSIGNMENT}(g) = \\ \quad \mathsf{mk\_type\_a\_statements}(\lambda(i,o,v).\,(g(i,o,v),v))$$

Example: In line 18 of the GCD circuit (figure 4.1) there is the following signal assignment

```
ready <= '0';
```

This signal assignment is represented by the following expression (see line with label 18 in figure 5.3)

$$\mathsf{SIGNAL\_ASSIGNMENT} \\ \quad (\lambda((a,b,start),(ready,result),(x,y,z)).\,(F,result))$$

Expanding the definition of $\mathsf{SIGNAL\_ASSIGNMENT}$ leads to the following equivalent expression

$$\mathsf{mk\_type\_a\_statement} \\ \quad (\lambda((a,b,start),(ready,result),(x,y,z)).\,((F,result),(x,y,z)))$$

Variable Assignments

Variable assignments are just the other way round: a single variable is altered whereas the other variables and the signals remain unchanged. Variable assignments can unambiguously be described by a function g of type $\iota \times \phi \to \phi$. The function $\mathsf{VARIABLE\_ASSIGNMENT}$ maps such functions to an expression of type $\mathsf{type\_a\_statement}$:

$$\vdash \mathsf{VARIABLE\_ASSIGNMENT}(g) = \\ \quad \mathsf{mk\_type\_a\_statement}(\lambda(i,o,v).\,(o,g(i,v)))$$

Null Statements

The null statement neither alters the output nor the variables. It is defined by:

$$\vdash \mathsf{NULL\_STATEMENT} = \\ \quad \mathsf{mk\_type\_a\_statement}(\lambda(i,o,v).\,(o,v))$$

Wait Statements

Wait statements are type $\mathcal{C}$ statements. There is exactly one wait statement position within a type $\mathcal{C}$ statement. Therefore, data type one is used to represent γ. The old control state and also the new control state are represented by type $(\mathsf{one})\mathsf{option}$. The formal definition is as follows:

$$\begin{array}{l} \vdash \mathsf{WAIT} = \\ \quad \mathsf{mk\_type\_c\_statement} \\ \quad\quad (\\ \quad\quad\quad (\lambda(i,o,v).\,(o,v,\mathsf{one})), \\ \quad\quad\quad (\lambda(i,o,v,c).\,(o,v,\mathsf{none})) \\ \quad\quad) \end{array}$$

The first function describes the behavior when starting from the beginning of the statement. The output and variable state are left unchanged and the process turns to the only possible control state named `one`.

The second function describes the way from some wait statement position within the statement. Since the set of wait statement positions is represented by the data type `one`, c is of type `one`, whose only value is the constant `one`. The output and variable state are left unchanged and the process turns to `none`, i.e. the end of the process.

5.4 Compound Statements

In $\mathcal{ABC}$-VHDL, there are three control structures for recursively combining basic statements to compound ones:

1. sequences,
2. if-then-else structures, and
3. while-loops

Control structures will be represented by functions that map tuples of statements and conditions to new (compound) statements. A sequences maps a pair of statements to a compound statement, an if-then-else structure maps a triple consisting of one condition and two statements to a compound structure, and a loop maps a pair consisting of a condition and a statement to a compound statements. The statements which are used as parts of control structures may either be basic statements or themselves be compound statements.

Since type $\mathcal{A}$, type $\mathcal{B}$ and type $\mathcal{C}$ statements have different logical types, it is not possible to represent each control structure by a single function but by a set of functions for each combination of statements. So there are 9 sequence functions and 9 if-then-else functions. However, there is only a single while-loop function since while loops are only allowed for type $\mathcal{C}$ statement bodies. The type of the compound statement can directly be derived from the types of its parts. Table 5.3 describes, how the type of the compound statement is derived from the types of its parts.

sequence

	$\mathcal{A}$	$\mathcal{B}$	$\mathcal{C}$
$\mathcal{A}$	$\mathcal{A}$	$\mathcal{B}$	$\mathcal{C}$
$\mathcal{B}$	$\mathcal{B}$	$\mathcal{B}$	$\mathcal{C}$
$\mathcal{C}$	$\mathcal{C}$	$\mathcal{C}$	$\mathcal{C}$

if-then-else

	$\mathcal{A}$	$\mathcal{B}$	$\mathcal{C}$
$\mathcal{A}$	$\mathcal{A}$	$\mathcal{B}$	$\mathcal{B}$
$\mathcal{B}$	$\mathcal{B}$	$\mathcal{B}$	$\mathcal{B}$
$\mathcal{C}$	$\mathcal{B}$	$\mathcal{B}$	$\mathcal{C}$

while

$\mathcal{C} \to \mathcal{B}$

Table 5.3: Construction Rules for Compound Statements

While loops are only allowed for type $\mathcal{C}$ statement bodies in order to avoid infinite loops. Both type $\mathcal{A}$ and type $\mathcal{B}$ statements may run from the beginning to the end position within one simulation cycle, i.e. without reaching a wait statement. While loops with type $\mathcal{A}$ or type $\mathcal{B}$ statement bodies therefore might be executed for an infinite number of times. In such a simulation, the next clock tick whould never be reached, and therefore such a circuit description whould not correspond to any real synchronous circuit.

Sequences

Sequence operators map a pair of statements to a compound statement. They are defined for all combinations of statement types. There names are SEQ_AA, SEQ_AB, SEQ_AC, SEQ_AA, etc. where the last two letters of the operator names indicate the types of the parameters. They are used in infix fashion.

γ represents the set of wait statements within a statement. When combining two statements by means of a sequence, the number of wait statement positions becomes the sum of the number of control statements of its parts. Type $\mathcal{A}$ statement do not contain wait statements. If both parts are type $\mathcal{A}$ statements, then the result also becomes a type $\mathcal{A}$ statement. If one of the parameters is a type $\mathcal{A}$ statement (with no wait statement) and the other statement is a type $\mathcal{B}$ or type $\mathcal{C}$ statement (with at least one wait statement), then the type for representing the wait statement set is inherited from the type $\mathcal{B}$ or type $\mathcal{C}$ statement. If both paramenters are type $\mathcal{B}$ or type $\mathcal{C}$ statements, and the set of wait statement positions is represented by γ^1 and γ^2, respectively, then the type of the compound statement becomes $\gamma^1 + \gamma^2$.

Example

Let x be a type $\mathcal{B}$ statement and y be a type $\mathcal{C}$ statement. Let

$$(\iota, \omega, \phi, \gamma^1)\mathsf{type\_b\_statement}$$

be the type of x and

$$(\iota, \omega, \phi, \gamma^2)\mathsf{type\_c\_statement}$$

be the type of y. The compound statement

$$(x\ \mathsf{SEQ\_BC}\ y)$$

is a type $\mathcal{C}$ statement (see table 5.3). According to appendix B, its type is

$$(\iota, \omega, \phi, \gamma^1 + \gamma^2)\mathsf{type\_c\_statement}$$

Therefore, the type of SEQ_BC is

$$(\iota, \omega, \phi, \gamma^1)\mathsf{type\_b\_statement} \to (\iota, \omega, \phi, \gamma^2)\mathsf{type\_c\_statement}$$
$$\to (\iota, \omega, \phi, \gamma^1 + \gamma^2)\mathsf{type\_c\_statement}$$

The semantics of the sequence operators are formalized within logic. The definitions describe, how the entire output and transition function of the compound statement is derived from the output and transition function of its parts. You find a complete table in appendix B.3.

Figure 5.4 informally sketches, how a type $\mathcal{B}$ and a type $\mathcal{C}$ statement are "bound" together. In simplified terms, the semantics of a sequence consisting of x and y is as follows:

1. When starting from the beginning or from one of the internal states of x, then first x is evaluated. If the control state of the result is an internal wait statement postition of x, then the result of the compound wait statement becomes the result produced so far. If the control state of the result is the end of x, then the output state and the variable state of the result and the current input are used to evaluate y from the beginning state. The result of the evaluation of y becomes the result of the sequence.

2. If the evaluation starts from one of the internal wait statements of y, then this result becomes the result of the entire sequence.

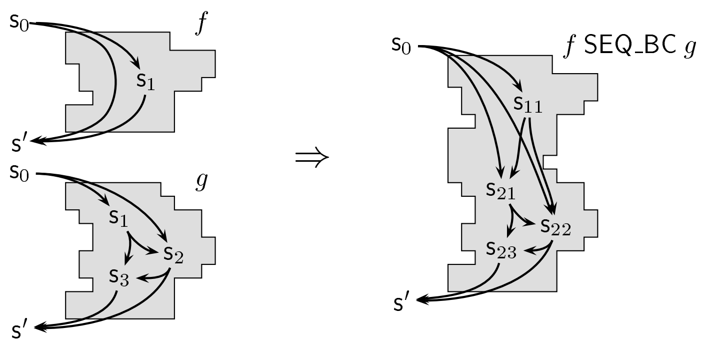


Figure 5.2: Sequence of a Type $\mathcal{B}$ and a Type $\mathcal{C}$ Statement

If-Then-Else Structures

If-then-else operators map two statements and a condition to a compound statement. They are defined for all combinations of statement types. There names are IF_THEN_ELSE_AA, IF_THEN_ELSE_AB, etc. where the last two letters of the operator names indicate the types of the statements.

As in sequences, the number of wait statement positions in if-then-else structures is the sum of its parts. So deriving γ for an if-then-else structure is done in the same way.

Example

Let c be a condition and x be a type $\mathcal{C}$ statement and y be a type $\mathcal{A}$ statement. Let

(ι, ϕ)condition

be the type of c,

$(\iota, \omega, \phi, \gamma^1)$type_c_statement

be the type of x and

$(\iota, \omega, \phi, \gamma^2)$type_a_statement be the type of y. The compound statement

$(\mathsf{IF\_THEN\_ELSE\_BC}\ c\ x\ y)$

is a type $\mathcal{B}$ statement (see table 5.3). According to appendix B, its type is

$(\iota, \omega, \phi, \gamma)\mathsf{type\_b\_statement}$

since the type of $\mathsf{SEQ\_BC}$ is

$(\iota, \phi)\mathsf{condition} \to (\iota, \omega, \phi, \gamma)\mathsf{type\_c\_statement} \to (\iota, \omega, \phi)\mathsf{type\_a\_statement} \to (\iota, \omega, \phi, \gamma)\mathsf{type\_b\_statement}$

An informal description of the semantics of if-then-else statements:

1. When starting from the beginning, then at first the condition c is evaluated. If the result of the condition is true, then the result of the compound statement is result of the application of the ouput and transition function of x. Otherwise y is applied.

2. If the evaluation starts from one of the internal wait statements of x, then this result becomes the result of the if-then-else structure.

3. If the evaluation starts from one of the internal wait statements of y, then this result becomes the result of the if-then-else structure.

While-Loops

The while statement operator maps a statement and a condition to a compound statement. While statements are only allowed for type $\mathcal{C}$ statement bodies. Other than type $\mathcal{A}$ and type $\mathcal{B}$ statements, type $\mathcal{C}$ statements cannot be evaluated all at once wihin one simulation cycle, but are always interrupted by a wait statement. Due to this restriction, while loops with infinite loops, i.e. runs without ever reaching a wait statement, are avoided.

The compound while statement is a type $\mathcal{B}$ statement, since:

1. there are wait statement positions in the body (type $\mathcal{C}$ statement)

2. starting from the beginning, the end may be reached emmediately (within one simulation cycle) if the condition fails

The while statement operator is named $\mathsf{WHILE\_C}$. Its type is as follows:

$(\iota, \phi)\mathsf{condition} \to (\iota, \omega, \phi, \gamma)\mathsf{type\_c\_statement}$
$\to (\iota, \omega, \phi, \gamma)\mathsf{type\_b\_statement}$

The semantics of a $(\mathsf{WHILE\_C}\ c\ x)$ is as follows:

1. When starting from the beginning, then the condition c is evaluated first. If the condition is fulfilled, then the body x is evaluated starting from its beginnning state. Since x is a type $\mathcal{C}$ statement, this leads to one fo the wait statement positions within x — the end cannot be reached emmediately. If the condition is false, then the end is reached emmediately.

2. When starting from one of the wait statement positions of the body, the evaluation of the output and transition function of the body may either lead to another wait statement position or to the end of the body. If another wait statement position is reached, then the evaluation of the while loop output and transition function is finished. If the end of the body is reached, then the condition is evaluated and according to the result either the end of the while-statement is reached or the evaluation is proceeded in the beginning. Due to x being a type $\mathcal{C}$ statement, the proceeding in the beginning will lead to a wait statement within the same evaluation cycle.

5.5 Statement Parts

As mentioned in section 4, the behaviour of processes is represented by an output and state transition function f of the corresponding automaton. In $\mathcal{ABC}$-VHDL, there are two kinds of processes:

1. type $\mathcal{A}$ processes and
2. type $\mathcal{C}$ processes

Type $\mathcal{A}$ processes

The statement part of a type $\mathcal{A}$ process is a (compound) type $\mathcal{A}$ statement with the following restrictions:

1. there are no variables
2. there are no if-then-else structures
3. there must be a sensisitivity list where *all* input signals are enlisted.

The ouput and transition function f of the corresponding automaton is nothing but the output and transition function of the type $\mathcal{A}$ statement. The function $\mathsf{CLOSE\_A}$ extracts this function from the type a statement. Let x, v and o be the type $\mathcal{A}$ statement in the body, the initial states of the variables and and the initial states of the outputs, respectively. The expression $(\mathsf{PROCESS\_A}\ (v, o)\ x)$ is used as an abbreviation for $\mathsf{automaton}(\mathsf{CLOSE\_A}\ x, (v, o))$.

Type $\mathcal{A}$ processes can be used to describe combinatorial circuits. Type $\mathcal{A}$ processes correspond to combinatorial circuit iff no variables are used and in all simulation cycles there is always at least one output signal assignment to each output signal.

Type $\mathcal{C}$ processes

The statement part of a type $\mathcal{C}$ process is a (compound) type $\mathcal{C}$ statement. Type $\mathcal{C}$ statements have no sensitivity list. Due to their wait statements, they are sensitive on changes of the clock signal.

Type $\mathcal{C}$ statements behave as follows: At time 0, the evaluation starts from the beginning and since the body is a type $\mathcal{C}$ statement, some wait statement position will be reached. The end cannot be reached emmediately. The initial control state, i.e. the beginning of the process will never be reached again.

In all further steps, the process jumps from one wait statement position to the next. Whenever the end is reached, the evaluation is proceeded in the beginning immediately within the same simulation cycle.

This very much resembles to a while loop with a condition that always holds. But others than while loops, the end is never reached. One could use the $\mathsf{WHILE\_C}$ loop with an always fulfilled condition to formalize this. The disadvantage of this approach is an extra control state at the end of the statement part. This extra control step whould be redundant, since it is unreachable. This has a negative impact as to verification and synthesis techniques (state exploration, state optimization, state encoding).

We therefore chose a formalization that differs from the $\mathsf{WHILE\_C}$ approach by not adding an this extra control state. The function deriving the functional behavior in terms of f from the type $\mathcal{C}$ statement body is named $\mathsf{CLOSE\_C}$.

Other than type $\mathcal{A}$ processes, the state of type $\mathcal{C}$ processes not only consists of variable state and output state, but there is also a control state. Let x be the the type $\mathcal{C}$ statement in the body. The initial variable state v and output state o are derived from the instantiations whereas the initial control state is always none. The expression $(\mathsf{PROCESS\_C}\ (v, o)\ x)$ is used as an abbreviation for $\mathsf{automaton}(\mathsf{CLOSE\_C}\, x, (v, o, \mathsf{none}))$.

```
                PROCESS_C
                  $((0,0,0),(\mathsf{F},0))$
                  (
[15,16,17]          WHILE_C ($\lambda((a,b,start),(x,y,z)).\ \not(start \neq \mathsf{T})$)WAIT SEQ_BC
[18]                SIGNAL_ASSIGNMENT($\lambda((a,b,start),(ready,result),(x,y,z)).\ (F,result)$) SEQ_AC
[19]                IF_THEN_ELSE_AA (mk_condition($\lambda((a,b,start),(x,y,z)).\ a < b$))
                    (
[20]                  VARIABLE_ASSIGNMENT($\lambda((a,b,start),(x,y,z)).\ (b,y,z)$) SEQ_AA
[21]                  VARIABLE_ASSIGNMENT($\lambda((a,b,start),(x,y,z)).\ (x,a,z)$)
                    )
                    (
[24]                  VARIABLE_ASSIGNMENT($\lambda((a,b,start),(x,y,z)).\ (a,y,z)$) SEQ_AA
[24]                  VARIABLE_ASSIGNMENT($\lambda((a,b,start),(x,y,z)).\ (x,b,z)$)
                    ) SEQ_AC
[26]                WHILE_C (mk_condition($\lambda((a,b,start),(x,y,z)).\ (y = 0)$))
                    (
[27]                  VARIABLE_ASSIGNMENT($\lambda((a,b,start),(x,y,z)).\ (x,y,x-y)$) SEQ_AC
[28]                  WAIT SEQ_CA
[29]                  VARIABLE_ASSIGNMENT($\lambda((a,b,start),(x,y,z)).\ (y,y,z)$) SEQ_AA
[30]                  VARIABLE_ASSIGNMENT($\lambda((a,b,start),(x,y,z)).\ (x,z,z)$)
                    ) SEQ_BC
[32]                SIGNAL_ASSIGNMENT($\lambda((a,b,start),(ready,result),(x,y,z)).\ (\mathsf{T},result)$) SEQ_AC
[33]                SIGNAL_ASSIGNMENT($\lambda((a,b,start),(ready,result),(x,y,z)).\ (ready,x)$) SEQ_AC
[34]                WAIT
                  )
```

Figure 5.3: Logical representation of the GCD circuit

Appendix A

Syntax of $\mathcal{ABC}$-VHDL

A.1 Conventions

Italic words are used to indicate syntactical expressions and type writer font is used for keywords. Syntactical definition is indicated by a `::=` symbol. $a \mid b$ is used to indicate an alternative, [a] is used to indicate an optional expression and $\{a\}$ is used to indicate, that the expression may be used n times in a series (with $n \in \{0, 1, \ldots\}$). All symbols except |,{, }, [,] indicate keywords. Some source text is part of the $\mathcal{ABC}$-VHDL language, if it can be matched with the syntactical expression *vhdl_programm* according to the syntax rules recursively describing it (see section A.3).

A.2 Basic Expressions

- *entity_identifier*, *architecture_identifier*, *component_identifier*, *signal_identifier*, *variable_identifier* and *library_identifier*:

  Arbitrary, user defined names used to indicate entities, architectures, components, signals, variable and libraries. Keywords are not allowed as identifiers. Identifiers of the same group and with the same scope must differ. Also variable identifiers and signal identifiers with the same scope must differ.

- *subtype_indication*:

  One of the data types defined in the type table (see section 2.3)

- *constant*, *infix_operator*, *prefix_operator* and *functor*:

  one of the constants, infix operators, prefix operators and functors defined in the constant table (see section 2.3)

A.3 Syntax Rules

architecture_body `::=`
  `architecture` *architecture_identifier* `of` *entity_identifier* `is` *architecture_declarative_part* `begin` {concurrent_statement} `end` [*architecture_identifier*] `;`

architecture_declarative_part `::=`
  {*block_declarative_item*}

association_element `::=`
  signal_identifier `=>` *signal_identifier* | *signal_identifier*

block_declarative_item `::=`
  signal_declaration | *component_declaration* |
  configuration_specification

component_declaration `::=`
  `component` *component_identifier* *port_clause* `end component ;`

component_instantiation_statement `::=`
  component_identifier `port map (` {*association_element* `,`}
  association_element `)`

concurrent_signal_assignment_statement `::=`
  signal_identifier `<=` *expression* `;`

concurrent_statement `::=`
  [*label_identifier* `:`] *concurrent_statement_unlabeled*

concurrent_statement_unlabeled `::=`
  process_statement | *component_instantiation_statement* |
  concurrent_signal_assignment_statement

configuration_specification `::=`
  `for` *instantiation_list* `:` *component_identifier* `use entitiy`
  library_identifier`.`*entity_identifier*`(`*architecture_identifier*`);`

entity_declaration `::=`
  `entity` *entity_identifier* `is` *port_clause* `end` [*entity_identifier*] `;`

expression `::=`
  variable_identifier |
  signal_identifier |
  constant |
  `(` *expression* `)` |
  expression *infix_operator* *expression* |
  prefix_operator *expression* |
  functor `(` {*expression* `,`} *expression* `)`

if_statement `::=`
  `if` *expression* `then` {*sequential_statement*} {`elseif` *expression* `then`}
  [`else` *sequential_statement*]

instantiation `::=`
  `:=` *expression*

instantiation_list `::=`
  {*label_identifier* `,`} *label_identifier* | `other` | `all`

interface_signal_declaration `::=`
  {*signal_identifier* ,} *signal_identifier* : [*mode*] *subtype_indication*
  [*instantiation*]

library_clause `::=`
  `library` {library_identifier ,} library_identifier ;

loop_statement `::=`
  `while` *expression* `loop` {sequential_statement} `end loop`

mode `::=`
  `in` | `out`

port_clause `::=`
  `port` ({*interface_signal_declaration* ;} *interface_signal_declaration*) ;

process_statement `::=`
  `process` [*sensitivity_list*] {*variable_declaration*} `begin`
  {*sequential_statement*} `end process` ;

sensitivity_list `::=`
  ({*signal_identifier* ,} *signal_identifier*)

sequential_statement `::=`
  wait_statement | *signal_assignment_statement* |
  variable_assignment_statement | *if_statement* | *loop_statement*

signal_assignment_statement `::=`
  signal_identifier `<=` *expression*

signal_declaration `::=`
  `signal` {*signal_identifier* ,} *signal_identifier* ; *subtype_indication*
  [instantiation] ;

variable_assignment_statement `::=`
  variable_identifier `:=` *expression*

variable_declaration `::=`
  `variable` {*variable_identifier* ,} *variable_identifier* :
  subtype_indication [*instantiation*]

vhdl_declaration `::=`
  entity_declaration | *architecture_body* | *library_clause*

vhdl_program `::=`
  {vhdl_declaration}

wait_statement `::=`
  `wait until` *signal_identifier* = `'1'` ;

Appendix B

Semantics of $\mathcal{ABC}$-VHDL

B.1 Basics

Type Constants

bool

one

num

$(\alpha)\mathsf{option}$

$\alpha + \beta$

Constants

$\mathsf{T} \ : \ \mathsf{bool}$

$\mathsf{F} \ : \ \mathsf{bool}$

$\mathsf{one} \ : \ \mathsf{one}$

$0 \ : \ \mathsf{num}$

$\mathsf{SUC} \ : \ \mathsf{num} \rightarrow \mathsf{num}$

$\mathsf{none} \ : \ (\alpha)\mathsf{option}$

$\mathsf{any} \ : \ \alpha \rightarrow (\alpha)\mathsf{option}$

$\mathsf{INL} \ : \ \alpha \rightarrow (\alpha + \beta)$

$\mathsf{INR} \ : \ \beta \rightarrow (\alpha + \beta)$

$\mathsf{PRIMREC\_bool} \ : \ \mathsf{bool} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$

$\mathsf{PRIMREC\_one} \ : \ \mathsf{one} \rightarrow \alpha \rightarrow \alpha$

$\mathsf{PRIMREC\_num} \ : \ \mathsf{num} \rightarrow \alpha \rightarrow (\alpha \rightarrow \mathsf{num} \rightarrow \alpha) \rightarrow \alpha$

$\mathsf{PRIMREC\_option} \;:\; (\alpha)\mathsf{option} \to \beta \to (\alpha \to \beta) \to \beta$

$\mathsf{PRIMREC\_sum} \;:\; (\alpha + \beta) \to (\alpha \to \gamma) \to (\beta \to \gamma) \to \gamma$

Theorems

$\vdash\ \forall a\, b.\ \exists^1 f.$
$\quad (f\,\mathsf{T} = a) \land$
$\quad (f\,\mathsf{F} = b)$

$\vdash\ (\forall a\, b.\ \mathsf{PRIMREC\_bool}\ \mathsf{T}\ a\ b = a) \land$
$\quad (\forall a\, b.\ \mathsf{PRIMREC\_bool}\ \mathsf{F}\ a\ b = b)$

$\vdash\ \forall a.\ \exists^1 f.$
$\quad f\,\mathsf{one} = a$

$\vdash\ \forall a.$
$\quad \mathsf{PRIMREC\_one}\,\mathsf{one}\ a = a$

$\vdash\ \forall a\, b.\ \exists^1 f.$
$\quad (f\,0 = a) \land$
$\quad (\forall c.\ f(\mathsf{SUC}\ c) = b\ (f\,c)\ c)$

$\vdash\ (\forall a\, b.\ \mathsf{PRIMREC\_num}\ 0\ a\ b = a) \land$
$\quad (\forall a\, b\, c.\ \mathsf{PRIMREC\_num}\ (\mathsf{SUC}\ c)\ a\ b = b\ (\mathsf{PRIMREC\_num}\ c\ a\ b)\ c)$

$\vdash\ \forall a\, b.\ \exists^1 f.$
$\quad (f\,\mathsf{none} = a) \land$
$\quad (\forall c.\ f(\mathsf{any}\ c) = b\ c)$

$\vdash\ (\forall a\, b.\ \mathsf{PRIMREC\_option}\ \mathsf{none}\ a\ b = a) \land$
$\quad (\forall a\, b\, c.\ \mathsf{PRIMREC\_option}\ (\mathsf{any}\ c)\ a\ b = b\ c)$

$\vdash\ \forall a\, b.\ \exists^1 f.$
$\quad (\forall c.\ f(\mathsf{INL}\ c) = a\ c) \land$
$\quad (\forall c.\ f(\mathsf{INR}\ c) = b\ c)$

$\vdash\ (\forall a\, b\, c.\ \mathsf{PRIMREC\_sum}\ (\mathsf{INL}\ c)\ a\ b = a\ c) \land$
$\quad (\forall a\, b\, c.\ \mathsf{PRIMREC\_sum}\ (\mathsf{INR}\ c)\ a\ b = b\ c)$

B.2 Type Abbreviations

Type Constants

(ι, ω, ϕ)type_a_statement

$(\iota, \omega, \phi, \gamma)$type_b_statement

$(\iota, \omega, \phi, \gamma)$type_c_statement

(ι, ϕ)condition

Constants

mk_type_a_statement :
$(\iota \times \omega \times \phi \to \omega \times \phi)$
$\to (\iota, \omega, \phi)$type_a_statement

dest_type_a_statement :
(ι, ω, ϕ)type_a_statement
$\to (\iota \times \omega \times \phi \to \omega \times \phi)$

mk_type_b_statement :
$(\iota \times \omega \times \phi \times (\gamma)$option $\to \omega \times \phi \times (\gamma)$option$)$
$\to (\iota, \omega, \phi, \gamma)$type_b_statement

dest_type_b_statement :
$(\iota, \omega, \phi, \gamma)$type_b_statement
$\to (\iota \times \omega \times \phi \times (\gamma)$option $\to \omega \times \phi \times (\gamma)$option$)$

mk_type_c_statement :
$(\iota \times \omega \times \phi \to \omega \times \phi \times \gamma) \ \times \ (\iota \times \omega \times \phi \times \gamma \to \omega \times \phi \times (\gamma)$option$)$
$\to (\iota, \omega, \phi, \gamma)$type_c_statement

dest_type_c_statement :
$(\iota, \omega, \phi, \gamma)$type_c_statement
$\to (\iota \times \omega \times \phi \to \omega \times \phi \times \gamma) \ \times \ (\iota \times \omega \times \phi \times \gamma \to \omega \times \phi \times (\gamma)$option$)$

Theorems

$\vdash \ \forall a. \ \exists^1 f.$
$\forall b. \ f($mk_type_a_statement $b) = a\, b$

$\vdash \ \forall x.$ mk_type_a_statement(dest_type_a_statement $x) = x$

$\vdash \ \forall x.$ dest_type_a_statement(mk_type_a_statement $x) = x$

$\vdash \ \forall a. \ \exists^1 f.$
$\forall b. \ f($mk_type_b_statement $b) = a\, b$

$\vdash \ \forall x.$ mk_type_b_statement(dest_type_b_statement $x) = x$

$\vdash \ \forall x.$ dest_type_b_statement(mk_type_b_statement $x) = x$

$\vdash \ \forall a.\ \exists^1 f.$
$\quad \forall b.\ f(\mathsf{mk\_type\_c\_statement}\ b) = a\ b$

$\vdash \ \forall x.\ \mathsf{mk\_type\_c\_statement}(\mathsf{dest\_type\_c\_statement}\ x) = x$

$\vdash \ \forall x.\ \mathsf{dest\_type\_c\_statement}(\mathsf{mk\_type\_c\_statement}\ x) = x$

$\vdash \ \forall a.\ \exists^1 f.$
$\quad \forall b.\ f(\mathsf{mk\_condition}\ b) = a\ b$

$\vdash \ \forall x.\ \mathsf{mk\_condition}(\mathsf{dest\_condition}\ x) = x$

$\vdash \ \forall x.\ \mathsf{dest\_condition}(\mathsf{mk\_condition}\ x) = x$

B.3 $\mathcal{ABC}$-VHDL Statements and Processes

Constants

negate_condition :
(ι, ϕ)condition $\rightarrow (\iota, \phi)$condition

SIGNAL_ASSIGNMENT :
$(\iota \times \omega \times \phi \rightarrow \omega)$
$\rightarrow (\iota, \omega, \phi)$type_a_statement

VARIABLE_ASSIGNMENT :
$(\iota \times \phi \rightarrow \phi)$
$\rightarrow (\iota, \omega, \phi)$type_a_statement

NULL_STATEMENT :
(ι, ω, ϕ)type_a_statement

WAIT :
$(\iota, \omega, \phi, \text{one})$type_c_statement

SEQ_AA :
(ι, ω, ϕ)type_a_statement $\rightarrow (\iota, \omega, \phi)$type_a_statement
$\rightarrow (\iota, \omega, \phi)$type_a_statement

SEQ_AB :
(ι, ω, ϕ)type_a_statement $\rightarrow (\iota, \omega, \phi, \gamma)$type_b_statement
$\rightarrow (\iota, \omega, \phi, \gamma)$type_b_statement

SEQ_AC :
(ι, ω, ϕ)type_a_statement $\rightarrow (\iota, \omega, \phi, \gamma)$type_c_statement
$\rightarrow (\iota, \omega, \phi, \gamma)$type_c_statement

SEQ_BA :
$(\iota, \omega, \phi, \gamma)$type_b_statement $\rightarrow (\iota, \omega, \phi)$type_a_statement
$\rightarrow (\iota, \omega, \phi, \gamma)$type_b_statement

SEQ_BB :
$(\iota, \omega, \phi, \gamma^1)$type_b_statement $\rightarrow (\iota, \omega, \phi, \gamma^2)$type_b_statement
$\rightarrow (\iota, \omega, \phi, \gamma^1 + \gamma^2)$type_b_statement

SEQ_BC :
$(\iota, \omega, \phi, \gamma^1)$type_b_statement $\rightarrow (\iota, \omega, \phi, \gamma^2)$type_c_statement
$\rightarrow (\iota, \omega, \phi, \gamma^1 + \gamma^2)$type_c_statement

SEQ_CA :
$(\iota, \omega, \phi, \gamma)$type_c_statement $\rightarrow (\iota, \omega, \phi)$type_a_statement
$\rightarrow (\iota, \omega, \phi, \gamma)$type_c_statement

SEQ_CB :
$(\iota, \omega, \phi, \gamma^1)$type_c_statement $\rightarrow (\iota, \omega, \phi, \gamma^2)$type_b_statement
$\rightarrow (\iota, \omega, \phi, \gamma^1 + \gamma^2)$type_c_statement

SEQ_CC :
$(\iota, \omega, \phi, \gamma^1)$type_c_statement $\rightarrow (\iota, \omega, \phi, \gamma^2)$type_c_statement
$\rightarrow (\iota, \omega, \phi, \gamma^1 + \gamma^2)$type_c_statement

IF_THEN_ELSE_AA :
$(\iota, \phi)condition$
$\rightarrow (\iota, \omega, \phi)$type_a_statement $\rightarrow (\iota, \omega, \phi)$type_a_statement
$\rightarrow (\iota, \omega, \phi)$type_a_statement

IF_THEN_ELSE_AB :
$(\iota, \phi)condition$
$\rightarrow (\iota, \omega, \phi)$type_a_statement $\rightarrow (\iota, \omega, \phi, \gamma)$type_b_statement
$\rightarrow (\iota, \omega, \phi, \gamma)$type_b_statement

IF_THEN_ELSE_AC :
$(\iota, \phi)condition$
$\rightarrow (\iota, \omega, \phi)$type_a_statement $\rightarrow (\iota, \omega, \phi, \gamma)$type_c_statement
$\rightarrow (\iota, \omega, \phi, \gamma)$type_b_statement

IF_THEN_ELSE_BA :
$(\iota, \phi)condition$
$\rightarrow (\iota, \omega, \phi, \gamma)$type_b_statement $\rightarrow (\iota, \omega, \phi)$type_a_statement
$\rightarrow (\iota, \omega, \phi, \gamma)$type_b_statement

IF_THEN_ELSE_BB :
$(\iota, \phi)condition$
$\rightarrow (\iota, \omega, \phi, \gamma^1)$type_b_statement $\rightarrow (\iota, \omega, \phi, \gamma^2)$type_b_statement
$\rightarrow (\iota, \omega, \phi, \gamma^1 + \gamma^2)$type_b_statement

IF_THEN_ELSE_BC :
$(\iota, \phi)condition$
$\rightarrow (\iota, \omega, \phi, \gamma^1)$type_b_statement $\rightarrow (\iota, \omega, \phi, \gamma^2)$type_c_statement
$\rightarrow (\iota, \omega, \phi, \gamma^1 + \gamma^2)$type_b_statement

IF_THEN_ELSE_CA :
$(\iota, \phi)condition$
$\rightarrow (\iota, \omega, \phi, \gamma)$type_c_statement $\rightarrow (\iota, \omega, \phi)$type_a_statement
$\rightarrow (\iota, \omega, \phi, \gamma)$type_b_statement

IF_THEN_ELSE_CB :
$(\iota, \phi)condition$
$\rightarrow (\iota, \omega, \phi, \gamma^2)$type_c_statement $\rightarrow (\iota, \omega, \phi, \gamma^1)$type_b_statement
$\rightarrow (\iota, \omega, \phi, \gamma^1 + \gamma^2)$type_b_statement

IF_THEN_ELSE_CC :
$(\iota, \phi)condition$
$\rightarrow (\iota, \omega, \phi, \gamma^1)$type_c_statement $\rightarrow (\iota, \omega, \phi, \gamma^2)$type_c_statement
$\rightarrow (\iota, \omega, \phi, \gamma^1 + \gamma^2)$type_c_statement

WHILE_C :
$(\iota, \phi)condition \rightarrow (\iota, \omega, \phi, \gamma)$type_c_statement
$\rightarrow (\iota, \omega, \phi, \gamma)$type_b_statement

$\mathsf{CLOSE\_A}$:
$$(\iota,\omega,\phi)\mathsf{type\_a\_statement} \to \iota\times\omega\times\phi \to \omega\times\omega\times\phi$$

$\mathsf{PROCESS\_A}$:
$$\omega\times\phi \to (\iota,\omega,\phi)\mathsf{type\_a\_statement} \to (\mathsf{num}\to\iota) \to \mathsf{num}\to\omega$$

$\mathsf{CLOSE\_C}$:
$$(\iota,\omega,\phi,\gamma)\mathsf{type\_c\_statement} \to \iota\times(\omega\times\phi)\times(\gamma)\mathit{option} \to \omega\times(\omega\times\phi)\times(\gamma)\mathit{option}$$

$\mathsf{PROCESS\_C}$:
$$\omega\times\phi \to (\iota,\omega,\phi,\gamma)\mathsf{type\_c\_statement} \to (\mathsf{num}\to\iota) \to (\mathsf{num}\to\omega)$$

Theorems

$$\vdash \forall cond.\ \mathsf{negate\_condition}\ cond = \mathsf{mk\_condition}(\neg \circ \mathsf{dest\_condition}\,cond)$$

$$\vdash \forall f.\ \mathsf{SIGNAL\_ASSIGNMENT}\ f = \mathsf{mk\_type\_a\_statement}(\lambda(si,so,sq).\ f((si,so,sq),sq))$$

$$\vdash \forall f.\ \mathsf{VARIABLE\_ASSIGNMENT}\ f = \mathsf{mk\_type\_a\_statement}(\lambda(si,so,sq).\ (so,f(si,sq)))$$

$$\vdash \mathsf{NULL\_STATEMENT} = \mathsf{mk\_type\_a\_statement}(\lambda(si,so,sq).\ (so,sq))$$

$$\vdash \mathsf{WAIT} = \mathsf{mk\_type\_c\_statement}\ (((\lambda(si,so,sq).\ (so,sq,\mathsf{one})),\ (\lambda(si,so,sq,ss).\ (so,sq,\mathsf{none})))$$

```
⊢ ∀pf, pg.
    (pf SEQ_AA pg) =
      mk_type_a_statement
        (λ(si, so, sq).
          let (so, sq) = dest_type_a_statement pf (si, so, sq) in
          let (so, sq) = dest_type_a_statement pg (si, so, sq) in
            (so, sq))
```

```
⊢ ∀pf, pg.
    (pf SEQ_AB pg) =
      mk_type_b_statement
        (λ(si, so, sq, ss).
          let (so, sq) = dest_type_a_statement pf (si, so, sq) in
          let (so, sq, ss) = dest_type_b_statement pg (si, so, sq, ss) in
            (so, sq, ss))
```

```
⊢ ∀pf, pg.
      (pf SEQ_AC pg) =
        mk_type_c_statement
          (let (pga, pgb) = dest_type_c_statement pg in
            (λ(si, so, sq).
              let (so, sq) = dest_type_a_statement pf (si, so, sq) in
              let (so, sq, ss) = pga(si, so, sq) in
                ((so, sq, ss), pgb)))

⊢ ∀pf, pg.
      (pf SEQ_BA pg) =
        mk_type_b_statement
          (λ(si, so, sq, ss).
            let (so, sq, ss) = dest_type_b_statement pf (si, so, sq, ss) in
              PRIMREC_option ss
                (let (so, sq) = dest_type_a_statement pg (si, so, sq) in
                  (so, sq, ss))
                (λdummy. so, sq, ss)))


⊢ ∀pf, pg.
      (pf SEQ_BB pg) =
        (let pfs = dest_type_b_statement pf in
        let pgs = dest_type_b_statement pg in
          mk_type_b_statement
            (λ(si, so, sq, ss12opt).
              PRIMREC_option ss12opt
                (let (so, sq, ss1opt) = pfs (si, so, sq, none) in
                  PRIMREC_option ss1opt
                    (let (so, sq, ss2opt) = pgs (si, so, sq, none) in
                      (so, sq, PRIMREC_option ss2opt none (λss2. any(INR ss2))))
                    (λss1. so, sq, any(INLss1)))
                (λss12. PRIMREC_sum ss12
                  (λss1.
                    let (so, sq, ss1opt) = pfs(si, so, sq, any ss1) in
                      PRIMREC_option ss1opt
                        (let (so, sq, ss2opt) = pgs(si, so, sq, none) in
                          (so, sq, PRIMREC_option ss2opt none (λss2. any(INR ss2))))
                        (λss1. so, sq, any(INL ss1)))
                  (λss2. let (so, sq, ss2opt) = pgs(si, so, sq, any ss2) in
                    (so, sq, PRIMREC_option ss2opt none (λss2. any(INR ss2))))))))
```

```
⊢ ∀pf, pg.
      (pf SEQ_BC pg) =
        mk_type_c_statement
          (let (pga, pgb) = dest_type_c_statement pg in
          let pfs = dest_type_b_statement pf in
            ((λ(si, so, sq).
              let (so, sq, ss1opt) = pfs (si, so, sq, none) in
                PRIMREC_option ss1opt
                  (let (so, sq, ss2) = pga (si, so, sq) in
                    so, sq, INRss2)
                  (λss1. so, sq, INL ss1)),
            (λ(si, so, sq, ss12).
              PRIMREC_sum ss12
                (λss1.
                  let(so, sq, ss1opt) = pfs(si, so, sq, any ss1) in
                    PRIMREC_option ss1opt
                      (let (so, sq, ss2) = pga(si, so, sq) in
                        (so, sq, any(INR ss2)))
                      (λss1. so, sq, any(INL ss1)))
                (λss2.
                  let (so, sq, ss2opt) = pgb(si, so, sq, ss2) in
                    (so, sq, PRIMREC_option ss2opt none (λss2. any(INR ss2)))))))

⊢ ∀pf, pg.
      (pf SEQ_CA pg) =
        mk_type_c_statement
          (let (pfa, pfb) = dest_type_c_statementpf in
            (pfa,
            (λ(si, so, sq, ss).
              let (so, sq, ssopt) = pfb(si, so, sq, ss) in
                PRIMREC_option ssopt
                  (let (so, sq) = dest_type_a_statementpg(si, so, sq) in
                    (so, sq, none))
                  (λss. so, sq, ssopt))))
```

```
⊢ ∀pf, pg.
    (pf SEQ_CB pg) =
      mk_type_c_statement
        (let (pfa, pfb) = dest_type_c_statement pf in
        let pgs = dest_type_b_statement pg in
          ((λ(si, so, sq).
            let (so, sq, ss1) = pfa (si, so, sq) in
              (so, sq, INLss1)),
          (λ(si, so, sq, ss12).
            PRIMREC_sum ss12
              (λss1.
                let (so, sq, ss1opt) = pfb(si, so, sq, ss1) in
                  PRIMREC_option ss1opt
                    (let (so, sq, ss2opt) = pgs(si, so, sq, none) in
                      (so, sq, PRIMREC_option ss2opt none (λss2. any(INR ss2))))
                    (λss1. (so, sq, any(INL ss1))))
              (λss2.
                let (so, sq, ss2opt) = pgs(si, so, sq, any ss2) in
                  (so, sq, PRIMREC_option ss2opt none (λss2. any(INR ss2)))))))

⊢ ∀pf, pg.
    (pf SEQ_CC pg) =
      mk_type_c_statement
        (let (pfa, pfb) = dest_type_c_statement pf in
        let (pga, pgb) = dest_type_c_statement pg in
          ((λ(si, so, sq).
            let (so, sq, ss1) = pfa(si, so, sq) in
              (so, sq, INL ss1)),
          (λ(si, so, sq, ss12).
            PRIMREC_sum ss12
              (λss1.
                let (so, sq, ss1opt) = pfb(si, so, sq, ss1) in
                  PRIMREC_option ss1opt
                    (let (so, sq, ss2) = pga(si, so, sq) in
                      (so, sq, any(INR ss2)))
                    (λss1. (so, sq, any(INL ss1))))
              (λss2.
                let (so, sq, ss2opt) = pgb(si, so, sq, ss2) in
                  (so, sq, PRIMREC_option ss2opt none (λss2. any(INR ss2)))))))

⊢ ∀cond, pf, pg.
    IF_THEN_ELSE_AA cond pf pg =
      mk_type_a_statement
        (λ(si, so, sq).
          (dest_condition cond (si, sq)) ⇒
            (dest_type_a_statement pf (si, so, sq)) |
            (dest_type_a_statement pg (si, so, sq)))
```

```
$\vdash$ $\forall cond, pf, pg.$
      IF_THEN_ELSE_AB $cond\, pf\, pg =$
        mk_type_b_statement
          $(\lambda(si, so, sq, ss).$
            let $pgs =$ dest_type_b_statement$pg$ in
              PRIMREC_option $ss$
                $(($dest_condition $cond\ (si, sq)) \Rightarrow$
                  (let $(so, sq) =$ dest_type_a_statement$pf(si, so, sq)$ in
                    $(so, sq,$ none$))\ |$
                  $(pgs(si, so, sq,$ none$)))$
                $(\lambda ss.\ pgs(si, so, sq,$ any$ss)))$

$\vdash$ $\forall cond, pf, pg.$
      IF_THEN_ELSE_AC $cond\, pf\, pg =$
        mk_type_b_statement
          (let $(pga, pgb) =$ dest_type_c_statement $pg$ in
            $\lambda(si, so, sq, ssopt).$
              PRIMREC_option $ssopt$
                $(($dest_condition$cond(si, sq)) \Rightarrow$
                  (let $(so, sq) =$ dest_type_a_statement $pf\ (si, so, sq)$ in
                    $(so, sq,$ none$))\ |$
                  (let $(so, sq, ss) = pga(si, so, sq)$ in
                    $(so, sq,$ any$ss)))$
                $(\lambda ss.\ pgb(si, so, sq, ss)))$

$\vdash$ $\forall cond, a, b.$
      IF_THEN_ELSE_BA $cond\ a\ b =$
        IF_THEN_ELSE_AB (negate_condition $cond)\ b\ a$

$\vdash$ $\forall cond, pf, pg.$
      IF_THEN_ELSE_BB $cond\, pf\, pg =$
        mk_type_b_statement
          $(\lambda(si, so, sq, ss12opt).$
            let $pfs =$ dest_type_b_statement $pf$ in
            let $pgs =$ dest_type_b_statement $pg$ in
              PRIMREC_option $ss12opt$
                $(($dest_condition$cond(si, sq)) \Rightarrow$
                  (let $(so, sq, ss1opt) = pfs(si, so, sq,$ none$)$ in
                    $(so, sq,$ PRIMREC_option$ss1opt$none$(\lambda ss1.$ any(INL $ss1))))\ |$
                  (let $(so, sq, ss2opt) = pgs(si, so, sq,$ none$)$ in
                    $(so, sq,$ PRIMREC_option$ss2opt$none$(\lambda ss2.$ any(INR $ss2)))))$
                $(\lambda ss12.$
                  PRIMREC_sum $ss12$
                    $(\lambda ss1.$
                      let $(so, sq, ss1opt) = pfs(si, so, sq,$ any$ss1)$ in
                        $(so, sq,$ PRIMREC_option $ss1opt$ none $(\lambda ss1.$ any(INL $ss1))))$
                    $(\lambda ss2.$
                      let $(so, sq, ss2opt) = pgs(si, so, sq,$ any$ss2)$ in
                        $(so, sq,$ PRIMREC_option $ss2opt$ none $(\lambda ss2.$ any(INR $ss2))))))$
```

```
⊢ ∀cond, pf, pg.
    IF_THEN_ELSE_BC cond pf pg =
      mk_type_b_statement
        (let (pga, pgb) = dest_type_c_statement pg in
          λ(si, so, sq, ss12opt).    let pfs = dest_type_b_statement pf in
              PRIMREC_option ss12opt
                ((dest_condition cond(si, sq)) ⇒
                  (let (so, sq, ss1opt) = pfs (si, so, sq, none) in
                    (so, sq, PRIMREC_option ss1opt none (λss1. any(INL ss1)))) |
                  (let (so, sq, ss2) = pga(si, so, sq) in
                    (so, sq, any(INR ss2))))
                (λss12.
                  PRIMREC_sum ss12
                    (λss1.
                      let (so, sq, ss1opt) = pfs(si, so, sq, any ss1) in
                        (so, sq, PRIMREC_option ss1opt none (λss1. any(INL ss1))))
                    (λss2.
                      let (so, sq, ss2opt) = pgb(si, so, sq, ss2) in
                        (so, sq, PRIMREC_option ss2opt none (λss2. any(INR ss2))))))

⊢ ∀cond, a, b.
    IF_THEN_ELSE_CA cond a b =
      IF_THEN_ELSE_AC (negate_condition cond) b a

⊢ ∀cond, a, b.
    IF_THEN_ELSE_CB cond a b =
      IF_THEN_ELSE_BC (negate_condition cond) b a

⊢ ∀cond, pf, pg.
    IF_THEN_ELSE_CC cond pf pg =
      mk_type_c_statement
        (let (pfa, pfb) = dest_type_c_statement pf in
        let (pga, pgb) = dest_type_c_statement pg in
          ((λ(si, so, sq).
            (dest_condition cond(si, sq)) ⇒
              (let (so, sq, ss1) = pfa(si, so, sq) in
                (so, sq, INL ss1))
              (let (so, sq, ss2) = pga(si, so, sq) in
                (so, sq, INR ss2))),
          (λ(si, so, sq, ss12).
            PRIMREC_sum ss12   (λss1.
                let (so, sq, ss1opt) = pfb(si, so, sq, ss1) in
                  (so, sq, PRIMREC_option ss1opt none(λss1. any(INL ss1))))
              (λss2.
                let (so, sq, ss2opt) = pgb(si, so, sq, ss2) in
                  (so, sq, PRIMREC_option ss2opt none(λss2. any(INR ss2)))))))
```

```
⊢ ∀$cond, pf$.
    WHILE_C $cond\, pf =$
      mk_type_b_statement
        (let $(pfa, pfb) =$ dest_type_c_statement $pf$ in
        let $conds =$ dest_condition $cond$ in
          $\lambda(si, so, sq, ssopt)$.
            PRIMREC_option $ssopt$
              $((conds(si, sq)) \Rightarrow$
                (let $(so, sq, ss) = pfa(si, so, sq)$ in
                  $(so, sq,$ any$ss)) \mid$
                $(so, sq,$ none$)$)
              $(\lambda ss.$
                let $(so, sq, ssopt) = pfb(si, so, sq, ss)$ in
                  PRIMREC_option $ssopt$
                    $((conds(si, sq)) \Rightarrow$
                      (let $(so, sq, ss) = pfa(si, so, sq)$in
                        $(so, sq,$ any$ss))$
                      $(so, sq,$ none$))(\lambda ss.\ (so, sq, ssopt))))$

⊢ ∀$pf$.
    CLOSE_C $pf =$
      (let $(pfa, pfb) =$ dest_type_c_statement $pf$ in
        $\lambda(si, (so', sq), ssopt)$.
          PRIMREC_option $ssopt$
            (let $(so, sq, ss) = pfa(si, so', sq)$ in
              $(so', (so, sq),$ any$ss))$
            $(\lambda ss.$
              let $(so, sq, ssopt) = pfb(si, so', sq, ss)$ in
                PRIMREC_option $ssopt$
                  (let $(so, sq, ss) = pfa(si, so, sq)$ in
                    $(so', (so, sq),$ any$ss))$
                  $(\lambda ss.\ (so', (so, sq),$ any$ss))))$

⊢ ∀$z0, pf$.
    PROCESS_C $z0\, pf =$
      automaton (CLOSE_C $pf, z0,$ none)

⊢ ∀$pf$.
    CLOSE_A $pf =$
      $(\lambda(si, so', sq)$.
        let $(so, sq) =$ dest_type_a_statement $pf\, (si, so', sq)$ in
          $(so', so, sq))$

⊢ ∀$z0, pf$.
    PROCESS_A $z0\, pf =$
      automaton(CLOSE_A $pf, z0$)
```

Appendix C

Standard $\mathcal{ABC}$-VHDL Type and Constant Semantics

The following tables only describe the default settings. Arbitrary mappings may be defined.

C.1 Translation for $\mathcal{ABC}$-VHDL Types

$\mathcal{ABC}$-VHDL	HOL
`boolean`	bool
`std_logic`	bool
`positive`	num

C.2 Translation for $\mathcal{ABC}$-VHDL Constants

$\mathcal{ABC}$-VHDL	HOL
`'0'`	F
`'1'`	T
`not`	$\neg$
`and`	$\wedge$
`or`	$\vee$
`mod`	MOD
`div`	DIV
`=`	$=$
`/=`	$\lambda(a,b).\neg(a=b)$
`-`	$-$
`+`	$+$
`<`	$<$
`>`	$>$
0,1,2,...	$0,1,2,\ldots$

Bibliography

[BGGH92] R. Boulton, A. Gordon, M. Gordon, J. Herbert, and J. van Tassel. Experiences with Embedding hardware description languages in HOL. In V. Stavridou, T.F. Melham, and R. Boute, editors, *Conference on Theorem Provers in Circuit Design*, IFIP Transactions A-10, pages 129–156. North-Holland, 1992.

[BrFK94] Peter T. Breuer, Luis Sanchez Fernandez, and Carlos Delgado Kloos. Clean formal semantics for VHDL. In *EDAC '94*, pages 641–647, Paris, France, 1994. IEEE Computer Society Press.

[DaJS93] W. Damm, B. Josko, and R. Schlör. A net-based semantics for VHDL. In Robert Werner, editor, *EURO–DAC '93*, pages 514–519, Hamburg, Germany, 1993. IEEE Computer Society Press.

[DeOd93] Alain Debreil and Philippe Oddo. Synchronous designs in VHDL. In Robert Werner, editor, *EURO–DAC '93*, pages 486–491, Hamburg, Germany, 1993. IEEE Computer Society Press.

[EiKu95] D. Eisenbiegler and R. Kumar. An automata theory dedicated towards formal circuit synthesis. In *Higher Order Logic Theorem Proving and Its Applications*, Aspen Grove, Utah, USA, September 1995. Springer.

[HaDa86] F.K. Hanna and N. Daeche. Specification and verification of digital systems using higher-order predicate logic. *IEE Proc. Pt. E*, 133(3):242–254, 1986.

[Melh88] F. Melham. Automating recursive type definitions in higher order logic. Technical Report 140, University of Cambridge Computer Laboratory, 1988.

[OlCo93] S. Olcoz and J.M. Colom. A petri net approach for the analysis of VHDL descriptions. In *CHARME93*, number 683 in Lecture Notes in Computer Science, pages 15–26, Arles,France, May 1993. Springer Verlag.

[ReKr93b] R. Reetz and T. Kropf. Hardwarebeschreibungssprachen und formale Verifikation. Technical Report SFB358-C2-4/93, Universität Karlsruhe, Institut für Rechnerentwurf und Fehlertoleranz, September 1993. http://goethe.ira.uka.de/hvg/techreports/SFB358-C2-4-93.ps.gz.

Index